Containers, microservices, and lambdas: the untold story

29 February 2024 - 6 min. read

AWS Lambda | Containers | Microservices

Containers, lambdas, and microservices are some of modern software development's most popular and important concepts. However, developers and architects often misunderstand or misuse these terms.

Sometimes, we are told that a project we're developing relies on a microservices architecture. In reality, it's only a bunch of tightly coupled little monoliths in containers.

In this article, we'll make things clear about microservices, the computational objects we can use to implement a service-oriented architecture, and the AWS-managed services that will ease our task.

"We use containers, so our system is microservices-oriented."

Unfortunately, things are not this easy. Microservices must adhere to strict rules and have clear definitions. Let's summarize and discuss them very briefly.

Microservices should be:

- **Tightly scoped:** microservices must implement a single function for the business logic, such as logging, user registration, price calculation
- **Loosely coupled**: a microservice should not depend on other microservices to implement an operation, and the availability of external services should not impact its availability.
- **Strongly encapsulated**: communication between microservices should be standard, no code should be shared, and their internals should not be accessible, even for data persistence.

- **Independently deployable**: deploying a new version of a microservice should not impact the availability and implementation of other services.
- **Independently scalable:** this comes as a consequence of the other requirements. If a service needs more computing resources, it should obtain them without depending on or impacting other services.

So, what about Containers? Let's see the principles of container design:

- **Single Concern**: a container addresses a single concern.
- **High Observability**: every container must implement APIs to observe its state and manage the application.
- **LifeCycle Conformance**: a container should be able to read events regarding the lifecycle of the platform that is running and react to the events.
- **Image Immutability**: a container is immutable; its code should not be specialized and changed between environments.
- **Process Disposability**: a containerized application should be ephemeral, so no data should be locally stored and used to persist its state.
- **Self-Containment**: libraries needed for the application are available inside the container's filesystem. The only dependency required is the kernel.
- **Runtime Confinement**: The runtime enforces the declared resource limits, so there will be no unexpected jeopardy in the platform.

As you may see, these definitions and concepts overlap in some points (and we didn't even mention Lambda!), but the devil is in the details: using a specific technology doesn't mean the architecture design will automatically have all the properties we expect.

In fact, **containers offer a way to package and execute software in an isolated and portable environment**. At the same time, **a microservices approach helps to divide a complex problem into multiple simpler, loosely coupled services** that communicate using APIs and can be managed by different teams.

With these key concepts in mind, let's dive deeper into the use of these technologies.

Benefits and challenges

Containers can help with scalability, portability, and resource efficiency but also require careful orchestration, monitoring, and security.

Lambdas can help with agility, cost-effectiveness, and scalability, but they also can be challenging to troubleshoot performance and problems due to the complexity of an implementation.

Microservices can help with modularity, flexibility, and fault tolerance but also introduce network latency, data consistency, and testing issues.

Here are some examples of containers and lambda that cannot be identified as microservices.

- A containerized WordPress (it's not tightly scoped)
- A lambda that relies entirely on another lambda output (it's not loosely coupled)
- Two containers that access the same database table (they're not encapsulated)

There are a lot of examples, and you surely can think of something you see in your everyday life.

We will not focus on whether a microservice-oriented architecture is better and for which use case because we aim to ease our lives while managing the infrastructures that will run our applications.

Remember that when we use containers and lambdas to implement a microservice strategy, we need to add at least three components to our architecture: a runtime, an orchestrator, and a discovery service.

As usual, AWS eases our lives because it offers different managed services to choose from; let's see them.

Compute

Compute is the most obvious part. We can use Amazon EC2, Fargate (for ECS or EKS), and AWS Lambda. EC2 seems counterintuitive, but microservices are about design, not technology.

Lambda and Fargate offer serverless computing platforms, so their billing is solely based on usage. They are the perfect companion for a microservices approach: if a Lambda-based microservice is rarely invoked, its billing will be marginal.

Using Fargate and Lambda also helps adapt to usage patterns: horizontal resource scaling using multiple small compute units optimizes scalability.

Orchestration

Orchestration is a critical component of a microservices architecture because it can automate the deployment, scaling, coordination, and fault-tolerance of microservices, managing also service lifecycle and execution flows.

ECS and EKS (on EC2 and Fargate) offer freedom of choice for the Container World and take away management effort; API Gateway and Step Functions help with the execution and integration of Lambdas and other AWS services.

AWS AppMesh can also be used to handle routing and authorization between services, adding an additional layer of observability.

Discovery Service

In a dynamic environment, accessing everything without relying on hardcoded names or IP addresses is mandatory (this requirement applies to every cloud environment).

Microservices should also be able to automatically notify their presence to a central registry that keeps track of what is where and facilitates communication.

The discovery service maintains a registry of the available service instances and provides a mechanism for service consumers to query and discover them, handling changes in the service instances due to scaling, failures, or upgrades.

EKS uses the built-in Kubernetes internal DNS, while ECS can rely on Amazon ECS Service Connect (tip: these two services can be used together).

Other things to keep in mind

To deploy microservices independently, you'll need pipelines. You will need to set up an efficient logging system to monitor containers. You'll need communication components (like queues, load balancers, and event triggers) to decouple your application.

All these additional components will surely increase the complexity of your architecture, so choose carefully: as we all know, sometimes we want to use a specific technology, bending the business case and design to our fantasy or curiosity. Choosing the design according to the desired technology, anyway, is dangerous and can lead a project to a spectacular failure because the wrong design will impact the availability, manageability, and, thus, the cost of owning the solution.

To Conclude

As always, there's no perfect solution: there are use cases that are a perfect fit for microservices, while, sometimes, breaking down every component in different services can only increase the overall complexity of our project: do you really need a user registration service for a local pet store e-commerce?

Starting with a monolith to speed up development time and ship a product faster could be a better approach than missing the right time to market a solution.

In which use case are you getting your hands on? Are you maintaining a legacy monolith, trying to break it down, or designing a new shiny system from scratch? Let us know in the comments!

(If you are considering refactoring a monolithic architecture into microservices, this article by my colleagues Alessandro and Simone can clarify things).

Good Luck with your infrastructure!

About Proud2beCloud

Proud2beCloud is a blog by beSharp, an Italian APN Premier Consulting Partner expert in designing, implementing, and managing complex Cloud infrastructures and advanced services on AWS. Before being writers, we are Cloud Experts working daily with AWS services since 2007. We are hungry readers, innovative builders, and gem-seekers. On Proud2beCloud, we regularly share our best AWS pro tips, configuration insights, in-depth news, tips&tricks, how-tos, and many other resources. Take part in the discussion!

Damiano Giorgi

Ex on-prem systems engineer, lazy and prone to automating boring tasks. In constant search of technological innovations and new exciting things to experience. And that's why I love Cloud Computing! At this moment, the only "hardware" I regularly dedicate myself to is that my bass; if you can't find me in the office or in the band room try at the pub or at some airport, then!
